

LINKED LISTS - OOP STYLE

RULE OF THREE

Problem Solving with Computers-II

C++

```
#include <iostream>
using namespace std;

int main(){
    cout<<"Hola Facebook!n";
    return 0;
}
```

GitHub



Questions to ask about any data structure:

- **What operations does the data structure support?**

A linked list supports the following operations:

1. Insert (a value to the head)
 2. Append (a value to the tail)
 3. Delete (a value)
 4. Search (for a value)
 5. Min
 6. Max
 7. Print all values
- **How do you implement each operation?**
 - **How fast is each operation?**

Linked List Abstract Data Type (ADT)

```
class LinkedList {
public:
    LinkedList();
    ~LinkedList();
    // other public methods

private:
    struct Node {
        int info;
        Node* next;
    };
    Node* head;
    Node* tail;
};
```

Memory Errors

- Memory Leak: Program does not free memory allocated on the heap.
- Segmentation Fault: Code tries to access an invalid memory location

RULE OF THREE

If a class overload one (or more) of the following methods, it should overload all three methods:

1. Destructor
2. Copy constructor
3. Copy assignment

The questions we ask are:

1. What is the behavior of these defaults?
2. What is the desired behavior ?
3. How should we over-ride these methods?

```
void test_append_0(){
    LinkedList l1;
    l1.append(10);
    l1.print();
}
```

Assume:

- * **Default destructor**
- * **Default copy constructor**
- * **Default copy assignment**

What is the result of running the above code?

- A. Compiler error
- B. Memory leak
- C. Segmentation fault
- D. None of the above

Why do we need to write a destructor for LinkedList?

- A. To free LinkedList objects
- B. To free Nodes in a LinkedList
- C. Both A and B
- D. None of the above

Behavior of default copy constructor

```
void test_copy_constructor() {  
    LinkedList l1;  
    l1.append(1);  
    l1.append(2);  
    LinkedList l2{l1};  
    // calls the copy c'tor  
    l1.print();  
    l2.print();  
}
```

Assume:

destructor: overloaded

copy constructor: default

What is the output?

- A. Compiler error
- B. Memory leak
- C. Segmentation fault
- D. All of the above
- E. None of the above

Behavior of default copy assignment

l1 : 1 -> 2 -> 5 -> null

```
void default_assignment_1(LinkedList& l1){  
    LinkedList l2;  
    l2 = l1;  
}
```

* What is the behavior of the default assignment operator?

Assume:

- * **Overloaded** destructor
- * **Default copy constructor**
- * **Default copy assignment**

Behavior of default copy assignment

```
void test_default_assignment_2(){
    LinkedList l1, l2;
    l1.append(1);
    l1.append(2)
    l2 = l1;
    l2.print()
}
```

What is the result of running the above code?

- A. Prints 1 , 2
- B. Segmentation fault
- C. Memory leak
- D. A &B
- E. A, B and C

Assume:

- * **Overloaded** destructor
- * **Default copy constructor**
- * **Default copy assignment**

Behavior of default copy assignment

```
void test_default_assignment_3(){
    LinkedList l1;
    l1.append(1);
    l1.append(2)
    LinkedList l2{l1};
    l2.append(10);
    l2.append(20);
    l2 = l1;
    l2.print()
}
```

What is the result of running the above code?

- A. Prints 1 , 2
- B. Segmentation fault
- C. Memory leak
- D. A &B
- E. A, B and C

Assume:

- * **Overloaded** destructor
- * **Overloaded** copy constructor
- * **Default** copy assignment

Overloading Binary Comparison Operators

We would like to be able to compare two objects of the class using the following operators

==

!=

and possibly others

```
void isEqual(const LinkedList & lst1, const LinkedList &lst2){  
    if(lst1 == lst2)  
        cout<<"Lists are equal"<<endl;  
    else  
        cout<<"Lists are not equal"<<endl;  
}
```

Overloading Binary Arithmetic Operators

We would like to be able to add two points as follows

```
LinkedList l1, l2;
```

```
//append nodes to l1 and l2;
```

```
LinkedList l3 = l1 + l2 ;
```

Overloading input/output stream

Wouldn't it be convenient if we could do this:

```
LinkedList list;  
cout<<list; //prints all the elements of list
```

Next time

- Binary Search Trees